Objectives:

(a) Describe the buffer overflow attack, determine what features of C make it possible, and identify who is responsible for memory management in C.
(b) Demonstrate the ability to craft simple buffer overflow exploits
(c) Explain how specific buffer overflow attacks work by describing stack operations.
(d) Analyze programs that submit input via the command line.

I. The Buffer Overflow Attack

1. Introduction  The very first major attack on DoD computer networks took place in February of 1998 and lasted for over a week. The hackers gained administrative (i.e., “root”) access on UNIX machines at 7 Air Force sites and 4 Navy sites, gaining access to logistical, administrative and accounting records. The method used in this early attack—a buffer overflow—has been used countless times ever since. Many famous attacks—the Morris Worm, the Code Red Worm, the SQL Slammer Worm, the Twilight Hack, Blaster, Conficker—used the buffer overflow as a primary attack vector. The recent Stuxnet worm used the buffer overflow as one of many attack vectors.

The buffer overflow attack is still exceedingly common. On January 3, 2014, the SANS Institute reported a newly discovered buffer overflow attack against the ubiquitous Linksys router. On January 9, 2014 a buffer overflow exploit was discovered in the “X Window” system that underpins many Linux desktops—although just discovered, this bug was waiting around to be discovered for the past 22 years! On January 15, 2014 (about two weeks ago!), a penetration testing firm announced the discovery of a zero-day flaw for executing a buffer overflow attack on a common SCADA system used in the US, the UK and Australia. A security researcher described the potential ramifications of this latter attack as “the stuff of modern-day nightmares.”

To be sure, the buffer overflow attack is not the only way to cripple a computer system. There are many other ways to attack, such as cross-site scripting, SQL injection, format string errors, and on and on. You may have learned in SI110 that the Department of Homeland Security worked together with the SANS Institute, Apple and Oracle back in 2011 to develop a list of the top 25 software vulnerabilities, and the “classic buffer overflow” came in third, behind SQL injection and OS command injection (cross-site scripting was 4th). The buffer overflow was the top vulnerability from 2000 through 2005, and has bounced around the top three spots ever since.

In February 2013, the security firm Sourcefire surveyed Common Vulnerability Scoring System (CVSS) data from 1988 to 2012, and found that buffer overflows were the most-often reported vulnerability. Of the vulnerabilities assigned a category of “high severity”, buffer overflows comprised over a third of the total. Security analyst Paul Roberts notes that “the stubborn staying power of buffer overflows for more than two decades – despite gallons of industry ink spilled on the problem – is dispiriting and has to get us thinking about what it is we’re doing wrong as an industry.”

2. In a Nutshell. The simple basis for the attack can be appreciated by examining the following section of C code:

```c
int k = 1000;
char my_stuff[512];
my_stuff[k] = 'A';
```
What happens if this code is executed? This array is only allotted 512 bytes; i.e., this array holds character variables my_stuff[0] through my_stuff[511]. The programmer who wrote the third line of code seems unaware that the last element of the array is my_stuff[511], since this third line of code assigns a value to the non-existent variable named my_stuff[1000]. When this code is executed, a byte of memory 488 bytes beyond the end of the array will be overwritten with the character 'A'.

This error will not be caught at compile-time. In a nutshell, the problem is that C compilers do not check for going beyond the bounds of an array.

This is a big concern because almost all major operating systems are written in C. Additionally, many popular applications are written in C.

You might be wondering: What exactly happens when the code above is run? The unfortunate answer is: Who knows? Perhaps nothing noticeable will occur. Perhaps disaster will occur.

Practice Problem

What feature of the C language makes a buffer overflow attack possible?

Solution:

3. Back to the Stack

Recall that when a program is to be executed, the operating system reserves a block of main memory for it.

The “text” segment holds the actual program (the machine language instructions which we can view as assembly-language instructions.) The memory allotted to the program in the text section does not change; it does not shrink or grow, since the program does not shrink or grow while it is being executed.

![Text segment for program 1]

Stack for program 1

The “stack” is the memory that the program has available to store information during execution. For example, the program’s variables are stored on the stack.
Let’s look at the program on the right, and examine the stack as it executes.

The program begins at the main function, and the variables that are used by the main function are placed on the stack. When the instruction pointer is at the location shown below on the right, the stack appears as on the left.

Recall that we keep track of the stack using the base pointer (ebp) which points to the bottom of the stack (specifically the memory location immediately following the bottom of the stack) and the stack pointer (esp) which points to the top of the stack. Each function gets to place its variables on the stack. The part of the stack that belongs to a function is called that function’s stack frame. So, the picture above depicts the current stack frame for the main function.

Now, the next instruction has us call the function named happy_times. The values of the arguments are placed on the stack in preparation for the function call. The stack, before the function call, now looks like this:
The function happy_times also has a variable (the array named alpha_code) and it needs to be allotted its own (separate) stack frame. But after happy_times are over, we will jump back to the main function. So, we still need to keep the stack frame for main undisturbed. Additionally, after happy_times are over, we need to resume program execution at the correct point (i.e., the point in main where we left off when we reached the function call).

So... what do we do?

We place the return address for the next instruction after the function call on the stack, and the old value of the base pointer on the stack, then we allot space for happy_times’ variable as shown below.

---

1 Of course happy times are never over at USNA....
Recall from last lecture that in a function call from main to another function, the stack will be organized as:

Note that our example conforms to this organization, as it must.

Now, suppose that the function happy_times, as part of its code (shown as “more code” above), prompts the midshipman to enter his alpha code. The function happy_times uses the character array named alpha_code to hold the value that the midshipman types in. We have seven bytes reserved on the stack for the alpha number (remember, we need the NULL terminator).

If all works well, all well and good. And everything always works well at USNA. Right?

Of course not!

Our midshipman was sleepy, and when he was prompted to enter his alpha code (which happens to be 151234) he dozed off for a micro-nap and accidentally entered:

```
1512344444444444444444444   <enter>
```

He entered a total of 25 characters. Think about this. What happens?

When the 25 characters are fed into the array alpha_code, the typed-in characters beyond the seventh will start overwriting memory!

It may be the case that the alpha code overwrites the return address.
Suppose this occurs. What will happen when function `happy_times` is finished executing? If the return address was indeed overwritten, then the return address will consist of some of the characters that were in the midst of the alpha string that was entered.

What will happen then? The instruction pointer will jump to some spurious address. And then... the program will most likely crash with a **segmentation fault**. A segmentation fault occurs when a program attempts to access memory outside the region of main memory that it has been allotted.

This sequence of events, if done intentionally, is called a **buffer overflow attack or a stack smashing attack!!!**

**Practice Problem**

Describe the mechanism by which a segmentation fault occurs.

**Solution:**

4. **The Buffer Overflow Attack on Steroids**

Our sleeping midshipman was not trying to do anything malicious—he just fell asleep like all midshipmen do. But how could this fundamental problem with C described above be exploited to do something truly evil?

So, now, Wacky Kim has placed a program into memory:
But how can Wacky Kim make use of this program?

Think about this: Suppose that when Wacky Kim types in his executable code, he takes care to carefully overwrite the return address, so that the four bytes that previously held the correct return address are changed to contain the address of alpha_code! In this case, the return address is the address of the start of the evil program that Wacky Kim has just placed in memory!

Consider the effect of this action. When function happy_times is done, the "return address" will be placed in the eip register. But the return address was adjusted to be the start of the executable program that has been surreptitiously placed in memory. So, Wacky Kim’s program will start executing.

In summary, Wacky Kim has placed his own program in memory and made it execute. Wacky Kim has executed a buffer overflow attack.

When examining the potential for a buffer overflow, the programmer should consider how a function's variables are placed on the stack. The first variable encountered is placed on the stack first, the second variable encountered is placed on the stack next (above the first variable) and so forth.

### Practice Problem

For the pawn function below, is it possible to overwrite the value you will get for your item with an amount of your choosing by overwriting the value variable on the stack during the scanf( ) call below? Explain.

```c
void pawn()
{
    char item[12];
    int value = 100;
    printf("What have you come to sell? ");
    scanf("%s", item);
}

int main()
{
    pawn();
}
```

Solution:
Practice Problem

When the **echo_string** function is called in **main** from the following code sample, the stack pictured below is created.

```c
#include<stdio.h>
void echo_string()
{
    int count;
    char entered_string[10];
    printf("Enter a string: ");
    scanf("%s", entered_string);
    for(count=0; count < 10; count=count+1)
    {
        printf("%s\n",entered_string);
    }
}
int main()
{
    echo_string();
}
```

Assuming there is no padding (extra spaces) when the frame is created. How many characters can be entered before the return address is overwritten?

Solution:

5. **A Possible Solution: Don't Use C!**

If this problem exists simply because C compilers do not check for going beyond the bounds of an array, an easy way to solve this problem would be to avoid using the C language altogether. In fact, more modern programming languages such as Java and C# will not allow a programmer to run beyond the bounds of an array. Why not simply abandon C and announce to the world: Problem Solved?

We cannot simply abandon C since too many C programs are in circulation. Moreover, programmers would not want to abandon C even if a magic wand could suddenly convert all C legacy code into Java programs! Recall from an earlier lecture that even today, most programmers are programming in C and prefer to program in C.

The C programming language is very popular because it executes quickly and it provides the programmer with a high level of control over the program. But with this nerd-power comes nerd-responsibility: Data integrity in C is the **programmer's responsibility**. If the responsibility for data integrity were taken away from the programmer and given to the compiler instead, the compiler would consistently and constantly check that we never run beyond the bounds of an array (which is good), but program execution would be much slower (which
is bad). Generally, users want their programs (whether they be operating systems, office software, application programs or games) to execute quickly. C executes quickly since the compiler does not verify data integrity. Yet, with the responsibility for data integrity resting on the programmer's shoulders, buffer overflow errors can occur if the programmer is not careful.

A good analogy is provided by Nick Rosasco: C is like a workbench with saws and power tools and high-voltage drops and spinning lathes all out in the open, without safeguards and protections. For a master craftsman who knows his job very well, this environment would be ideal for productive work, with the understanding that the craftsman has to be responsible for his safety. For the novice, this environment would be very dangerous.

Conversely, a workbench that required the user to constantly interact with multi-level interlocked protection mechanisms and cumbersome safety features would be much safer for the novice, but would drive the skilled craftsman insane. As with work benches, so with programming languages: The intentional lack of safety in C translates into greater flexibility and improved performance... and risk.

In order for you to write your own buffer overflow attacks, we have to add a little bit to your C repertoire. For today, we have to cover command line arguments and the `exit` command. It’ll be fun.

II. More Fun with C

1. **Command Line Arguments.** Up to this point, we have written the first line of the function `main` as

   ```c
   int main()
   ```

   However, `main` is a function that we can pass arguments to. As we already know, `main` is special, and passing arguments to the `main` function also takes place in a special way.

   The main function is more formally written as

   ```c
   int main (int argc, char *argv[]) 
   ```

   The parameter `argc` contains the number of arguments passed to `main` and the variable `argv` is an array of strings with each argument passed stored in one of the array locations.

   First, let’s get a little bit comfortable with this notation. If we type in the following program:

   ```c
   #include <stdio.h>
   int main( int argc, char *argv[] )
   {
     int i;
     printf("Arguments to this program, on the command-line:\n");
     for( i = 0; i < argc; i = i + 1 )
       printf("%s\n", argv[i]);
   }
   ```

   then, when executing it we would see the output below:

   ```plaintext
   midshipman@EC310:~ $ ./a.out
   Arguments to this program, on the command-line:
   ./a.out
   midshipman@EC310:~ $ 
   ```
Here is what is happening. When you execute a C program, the operating system counts the total number of separate items entered, and places that integer in the variable `argc`. Each separate item you entered is placed, as a string, one-by-one, in the array of strings `argv`.

So, if I was to type: `./a.out one 2 3.45 who?`

Then: `./a.out one 2 3.45 who?`


and what is the value of `argc`? The answer: 5.

**Practice Problem**

For the following program invocation: `midshipman@EC310 ~$ ./a.out wait 8 mate`

A) What is the value of `argc`?
B) What is the value of `argv[1]`?
C) What is the data type of `argv[2]`?

Solution: (a) (b) (c)

**Practice Problem**

Pertaining to taking in command line arguments for a program, choose the best description for `argc`.

(A) holds the number of command line arguments excluding the program name.
(B) holds the total number of command line arguments available to the program.
(C) holds the number of integer variables entered at the command line before the program begins.
(D) None of the above.

Solution:
Practice Problem

In the following sentence, circle the correct choices.

argv is a(n) array / index / stack used to store each command line parameter / index / argument in a binary / string / numeric format.

Solution:

2. The exit statement. Sometimes we would like to intentionally terminate a program “gracefully” (instead of letting the program crash and burn). This can be accomplished with an exit statement. When using the exit statement, we must add the directive: #include<stdio.h>. An example:

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    float x, y;
    printf( "This program divides x by y \n" );
    printf( "Enter x and y: " );
    scanf( "%f %f", &x, &y );
    if( y == 0 )
    {
        printf( "Divide by 0!\n" );
        exit(1); //For us, it doesn’t matter what number we use
    }
    else
    {
        printf( "x/y is %f\n" , x/y);
    }
}
```
Problems

1. What features of the C language make a buffer overflow attack possible?

2. Answer the following questions concerning how a program is stored in memory during its execution.
   (a) Which segment of memory has contents that remain unchanged during program execution?
   (b) Does the programmer have complete control over how the stack is organized?
   (c) What is the relationship between the order in which variables appear in a function and the order in which these same variables are stored in the function's stack frame?
   (d) What important registers are used to define the boundaries of a stack frame?
   (e) Suppose main calls a function named fun. After all the commands of fun have executed, how does the program know to continue at the exact location in main where it left off?
   (f) Is a source code file permitted to have more than one function?
   (g) If your answer to (f) was "no", explain why that is the case. If your answer to (e) was "yes", explain how the operating system knows where to begin executing your program if the source code file contains multiple functions.

3. Segmentation Fault Carefully enter the following program using nano. Notice that the program has no blank lines.

```c
#include<stdio.h>
void happy_times( int x , int y )
{
    char alpha_code[ 7 ];
    printf("Enter your alpha code:" );
    scanf( "%s", alpha_code );
    printf("Your alpha code is: %s\n\n", alpha_code );
}
int main( )
{
    int a = 77;
    int b = 21;
    happy_times( a , b);
}
```

Execute the program entering just the numeric portion of your alpha code. You should see something like this:

```
midshipman@EC310:~/work $ ./a.out
Enter your alpha code:151234
Your alpha code is: 151234
midshipman@EC310:~/work $ 
```
Now, rerun the program entering a ridiculously long alpha code. You should see a segmentation fault:

```
midshipman@EC310:~/work $ ./a.out
Enter your alpha code:151111111111111111111111111111111111
Your alpha code is: 151111111111111111111111111111111111
Segmentation fault
```

Recall that a segmentation fault occurs if a program attempts to run beyond the boundaries of main memory that the operating system has allotted the program. In this homework problem we will explore in depth the cause of this segmentation fault.

Let's run our program (which I've named `happy.c`) by entering:

```
gcc -g happy.c
gdb -q ./a.out
set dis intel
list main
break 13
run
nexti
nexti
nexti
nexti
```

Exactly four `nexti`'s

If you now enter

```
i r eip
```

you should confirm that the next instruction that will execute is the instruction at address \texttt{0x804819}. If you now enter

```
disass main
```

you should verify that the very next instruction is the function call. See the screen capture on the next page.
The important point of all this is to note that you are still in \texttt{main} (but just barely!).

Recalling the generic picture of the stack, and noting that we have not yet arrived at the function call, the stack should consist just of \texttt{main}'s variables and the function's arguments.

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{function's arguments} & \textbf{main's variables} \\
\hline
\end{tabular}
\end{center}

(a) Our goal is to locate \texttt{main}'s variables and the function's arguments on the stack. Recall that \texttt{main}'s variables (\texttt{a} and \texttt{b}) will be stored in binary, which we can read as hexadecimal numbers. Convert the values of \texttt{a} and \texttt{b} to hexadecimal and write these values below as eight hexadecimal digits (recall that integers are stored as four bytes, and four bytes equates to eight hexadecimal digits):

Note: For Parts (b) – (i) you will fill in the table which begins on page 209.

(b) Examine the value of the stack pointer (\texttt{i r esp}) and the base pointer (\texttt{i r ebp}). Fill in the values in the table below, showing where the base pointer (label as \texttt{EBP-main}) and stack pointer (label as \texttt{ESP-main}) are pointing to.
(c) Look at 40 bytes starting at the stack pointer by entering

```
x/40xb $esp
```

You should see:

This is the contents of memory location **0xbffff800**

This is the contents of memory location **0xbffff801**

This is the contents of memory location **0xbffff802**

<table>
<thead>
<tr>
<th>(gdb) x/40b $esp</th>
<th>0x7e</th>
<th>0x00</th>
<th>0x00</th>
<th>0x7e</th>
<th>0x15</th>
<th>0x00</th>
<th>0x00</th>
<th>0x00</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbffff800:</td>
<td>0x4d</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x15</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0xbffff808:</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0xbffff810:</td>
<td>0x15</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x4d</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0xbffff818:</td>
<td>0x78</td>
<td>0xf8</td>
<td>0xf8</td>
<td>0xb7</td>
<td>0xb7</td>
<td>0xe8</td>
<td>0xe8</td>
<td>0xe8</td>
</tr>
<tr>
<td>0xbffff820:</td>
<td>0x01</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0xa4</td>
<td>0xf8</td>
<td>0xff</td>
<td>0xbf</td>
</tr>
</tbody>
</table>

Locate *main*’s variables and the function’s arguments on the stack. Fill in the table, annotating the locations of these four values. Label these as *(main variable: a)*, *(main variable b)*, *(function argument: x)* and *(function argument: y)*.

(d) Now enter

```
break 2
continue
nexti
```

The program is now at the point where the old value of the base pointer and the correct return address have been placed on the stack.

<table>
<thead>
<tr>
<th>old value of ebp</th>
<th>Return address</th>
<th>function’s arguments</th>
<th>main’s variables</th>
</tr>
</thead>
</table>

What should be stored as the correct return address? (Hint: enter *disass main* and determine the address of the next instruction after the function call.)

What should be the saved value of the base pointer?

(e) Examine the value of the stack pointer *(i r esp)*. Fill in the values in the table below, showing the stack pointer's location *(label as ESP-main-revised)*.

(f) Look at 40 bytes starting at the stack pointer by entering

```
x/40xb $esp
```

Locate the saved value of the base pointer and the return address on the stack. Fill in the table, annotating the locations of these two items. Label these as *(saved base pointer)* and *(return address)*.
(g) Now enter

```
break 8
continue
```

When prompted to enter your alpha code, enter: AAAAAA

Examine the value of the stack pointer (\texttt{i r esp}) and the base pointer (\texttt{i r ebp}). Fill in the values in the table below, showing where the base pointer (label as \texttt{EBP-happy\_times}) and stack pointer (label as \texttt{ESP-happy\_times}) are pointing to.

<table>
<thead>
<tr>
<th>function's variables</th>
<th>old value of ebp</th>
<th>Return address</th>
<th>function's arguments</th>
<th>main's variables</th>
</tr>
</thead>
</table>

(h) Locate your alpha code in the stack frame for \texttt{happy\_times}. Do this by examining 40 bytes starting at the stack pointer. Note that the capital letter A is equivalent to hexadecimal 0x41. Fill in the table, annotating the location of the string \texttt{alpha\_code}. Note that the NULL that terminates the string is part of the string.

(i) Now, examine your memory drawing. How many characters would you have had to enter for your alpha code before you start to overwrite the saved value of the base pointer (remember that the NULL is automatically added)?

Overwriting the saved value of the base pointer will (almost always) cause a segmentation fault, because the program will attempt to restore the stack to a location in memory outside the region of main memory given to the program.

(j) Exit the debugger (by entering \texttt{quit}) and run your program by entering \texttt{./a.out}. Enter an alpha code of size equal to the number of characters you calculated in part (i). Did you get a segmentation fault? (You should have!)

(k) Enter an alpha code of size \textbf{one less than} the number of characters you calculated in part (i). Did you get a segmentation fault? (You should not have.)
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFFFF7CD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7CF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7D0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7D1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7D2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7D3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7D4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7D5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7D6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7D7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7D8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7D9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7DA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7DB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7DC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7DD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7DE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7DF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7E0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7E1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7E2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7E3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7E4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7E5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7E6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7E7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7E8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7E9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7EA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7EB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7EC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7ED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7EE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7EF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7F0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7F1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7F3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7F4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7F5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7F6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7F7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7F8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7F9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7FA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7FB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF7FC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. Given the following code snippet:

```c
char first_name[6] = "Alice";
strcpy(first_name, "Alexander");
```

(a) Will the C compiler state that there is an error?

(b) What potentially dangerous situation occurs because of the snippet above?

(c) What is the minimum size necessary for the array first_name to prevent this error?

(d) There are at least two ways to change the above code to prevent the above error from happening. Describe one.

5. When the `greetings` function is called in `main` from the following code sample the stack pictured below is created.

```c
#include<stdio.h>
void greetings()
{
    int name_len = 15;
    char name[name_len];
    int year = 2014;

    printf("Enter your name: ");
    scanf("%s", name);
    printf("Hello: %s! The current year is %d.\n", name, year);
}

int main()
{
    greetings();
}
```

(a) Assuming there is no padding (extra spaces) when the frame is created, how many characters must the user enter to overwrite only the first byte of the return address?

(b) Is it possible to change the value of `year` by performing a buffer overflow attack? Why or why not?